Chapter 3

Scheduling

There cannot be a crisis next week. My schedule is already full. [Henry Kissinger]

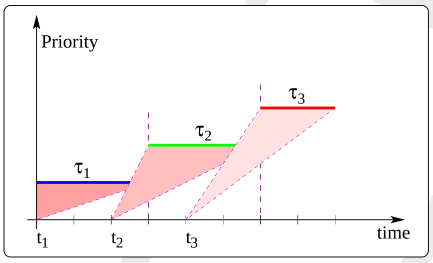
Concepts introduced: Deadlines, feasability, schedulability, precedence constraints, resource constraints, process utilization factor, RMS, EDF, mutual exclusion, critical sections, semaphores, priority inheritance and ceiling protocols.

IN CHAPTER 2, a structure of hard real-time systems is outlined in which special purpose hardware is used to run a set of fixed duration tasks. Because of the *hard* requirement for the RT systems, we do not want to use probabilistic ordering (scheduling) of tasks. Instead we prefer to use scheduling algorithms which precisely determine at which time each task executes. We begin by exploring the underlying structure of *tasks* in the hard RT computing environment.

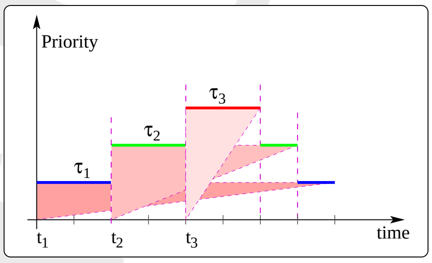
3.1 Task scheduling concepts

The underlying OS is responsible for scheduling the tasks, and the part of the OS that does this is termed the scheduler. A preemptive scheduler can preempt (interrupt) other tasks if a higher priority task needs to be scheduled. This may improve the responsiveness of the system, but complicates the scheduling.

Consider a situation where we have a set of three tasks $\{\tau_1, \tau_2, \tau_3\}$ with a relative priority or importance $p_i = \text{Priority}(\tau_i)$ such that $p_1 < p_2 < p_3$. If the three tasks took 3, 3 and 2 msecs respectively, and they were scheduled to run at times $t_1 = 0$, $t_2 = 2$ and $t_3 = 4$, then a non-preemptive scheduler would run the tasks as seen in Figure 3.1(a), where τ_2 and τ_3 are delayed from starting.



(a) Non-preemptive



(b) Preemptive

Figure 3.1: Scheduler preemption

A preemptive scheduler as seen in Figure 3.1(b), will not delay the start of the tasks, and τ_3 (the highest priority task) completes as soon as possible. Note that task τ_2 does not complete until after τ_3, and also that if there was some other constraint on τ_1 (for example: τ_1 must complete less than 5 msecs after it starts), then the preemptive scheduler would not meet this constraint.

Definition 12 *A schedule is **feasible** if all tasks can be completed while satisfying the given constraints.*

Definition 13 *A task set is **schedulable** if there is at least one scheduling algorithm which produces a feasible schedule.*

3.1.1 Scheduling constraints

The constraints may be timing, precedence or resource constraints.

For timing constraints, the principal one is that a task should meet some deadline. In the case of RT systems in this book, this will be a *hard* constraint. We associate with each task τ_i an arrival or activation time a_i, and a computation time c_i which is the time needed to perform the task if there was no preemption. We assume a priority p_i for each task, and start and finish times for each task are s_i and f_i respectively.

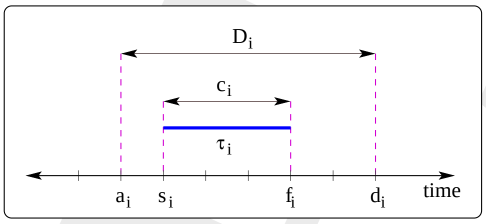


Figure 3.2: Times associated with a task

Definition 14 *If a task t_i needs to finish before some time d_i, then this is called a **deadline**. A **relative deadline** D_i for the task is $D_i = d_i - a_i$.*

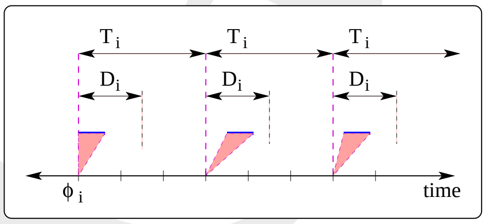


Figure 3.3: Periodic tasks

Definition 15 *A **periodic** task is one that is regularly activated at a constant rate. Its period is T_i, and the time of first activation (its **phase**) is ϕ_i.*

It is OK if the start time s_i of a task is delayed, as long as the finish time f_i is before the expiry of the (relative) deadline.

Precedence constraints are those that specify the ordering of tasks. Sometimes a task may not be activated until after some other task has completed. We use the notation $\tau_a \prec \tau_b$ to indicate that τ_a precedes τ_b, and the notation $\tau_a \rightarrow \tau_b$ to indicate that τ_a immediately precedes τ_b. A precedence graph can be used to show the precedence relation between tasks.

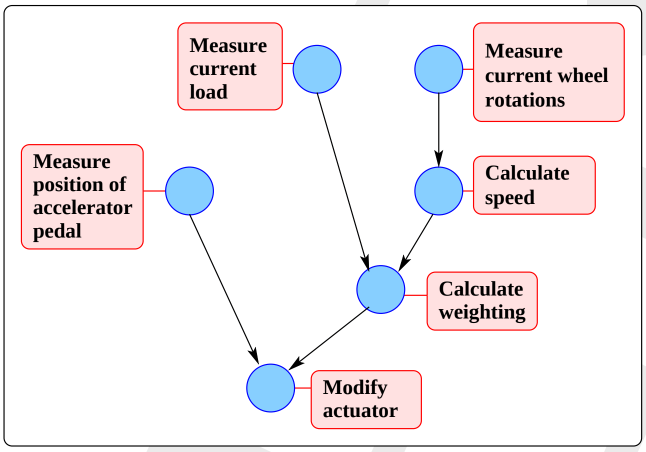


Figure 3.4: Precedence graph for car control system

> For example, consider a vehicle system, which measures the speed of the car, and the load on the engine to calculate a weighting factor, which is then used to modify the adjustment of the carburettor depending on the position of the accelerator pedal.

Each of the items introduced in this description may be implemented using a set of tasks, but we cannot (for example) modify the carburettor if we have not finished calculating the weighting factor. In Figure 3.4, we see a possible precedence graph for this system.

The last form of critical constraint is a *resource* constraint, which may be some variable or device or some other structure in the system. Resources only become *critical* resource constraints when they are shared with other tasks.

An *exclusive* resource is one which may require exclusion of all other tasks when the resource is accessed. This is called *mutual exclusion*, and operating systems normally provide synchronization mechanisms to assist tasks to provide mutually exclusive access to a resource. The code which requires this mutually exclusive access is termed a *critical section* (CS).

The most common mechanism for this purpose is called the semaphore, where a semaphore variable s_i is used to control access to an associated CS_i. The two functions on the semaphore are:

1. $\mathrm{wait}(s_i)$; will block if $s_i = 0$, but otherwise it will pass through, setting s_i to 0.
2. $\mathrm{signal}(s_i)$; sets s_i to 1.

The OS manages blocking and unblocking of tasks by keeping queues for each type of task. A task waiting for an exclusive resource is blocked on that resource, and tasks blocked on the same resource are kept in a wait queue associated with the semaphore protecting the resource.

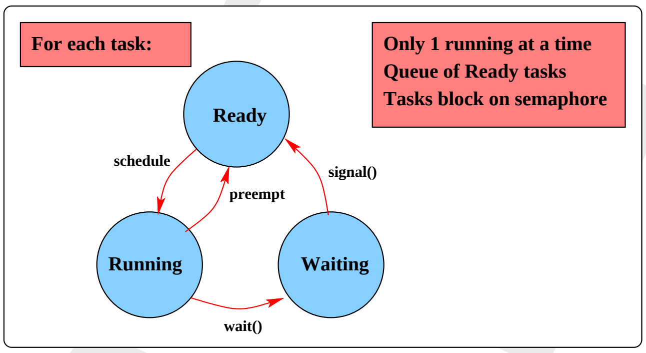


Figure 3.5: States of the tasks

Figure 3.5 shows the states of the tasks waiting for access to resources. A task in the running state executing `wait(s)` on a locked semaphore enters the waiting state. When a task currently using the resource executes `signal(s)`, the semaphore is released. When a task leaves its waiting state (because the semaphore has been released) it goes into the ready state, to be scheduled later.

3.2 The scheduling problem

The general scheduling problem is NP-complete. There is a non-deterministic Turing Machine TM and a polynomial in one variable $p(n)$ such that for each problem instance of size n, TM determines if there exists a schedule and if so outputs one in at most $p(n)$ steps. Any non-deterministic polynomial time problem can be transformed in deterministic polynomial time to the general scheduling problem, and only exponential time deterministic algorithms are known.

Hence we must find imperfect but efficient solutions to scheduling problems. A great variety of algorithms exist, with various assumptions, and with different complexities.

3.2.1 RMS - Rate monotonic scheduling

Much of the analysis of scheduling algorithms is based on statistical analysis of the behaviour of various queuing strategies. By contrast, in a hard real-time environment we are only interested in absolute results, and to this end we make assumptions about the environment in which RMS operates.

In RMS [LL73], we assume a set of tasks $\{\tau_1, \ldots, \tau_m\}$ with periods $T_1, \ldots, T_m$, $\phi_i = 0$ and $D_i = T_i$ for each task. We allow preemption, there is only a single processor, and we have no precedence constraints. From the Liu article:

(A1) The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.

(A2) Deadlines consist of run-ability constraints only - i.e. each task must be completed before the next request for it occurs.

(A3) The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.

(A4) Run-time for each task is constant for that task and does not vary with time.

(A5) Any nonperiodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

The RMS approach assigns higher priorities to tasks with shorter periods. If higher priority tasks arrive, they can preempt the currently running task. RMS is a static scheduling policy.

	τ_1	τ_2	τ_3
C_i	1	2	3
T_i	4	6	10

Table 3.1: Task set for scheduling

For the periodic tasks given in Table 3.1, Figure 3.6 shows an RMS schedule.

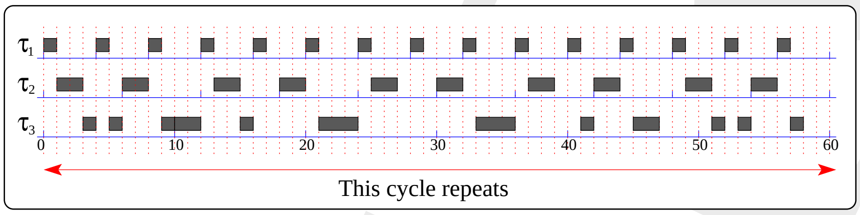


Figure 3.6: RMS schedule for tasks

As you can see, RMS is a very simple algorithm, but even so, if a set of tasks with the aformentioned constraints is not schedulable with RMS, then no static scheme can schedule them.

Definition 16 *The* ***processor utilization factor*** U *is the fraction of processor time spent in the task set:*

$$\mathrm{U} = \sum_{i=1}^{m} \frac{C_i}{T_i}$$

If this factor is *greater* than 1 then of course, the task set can not be scheduled. However if $\mathrm{U} \leq 1$ then it is possible that it may be RMS-schedulable. If a particular set of tasks has a feasible RMS schedule, and any increase of the runtime of any task would render the particular set infeasible, then the processor is said to be fully utilized.

Definition 17 *The* ***least upper bound*** *of* $\mathrm{U_{lub}}$ *is the minimum of the* U *over all sets of tasks that fully utilize the processor.*

If $\mathrm{U} \leq \mathrm{U_{lub}}$, then the set of tasks is guaranteed to be schedulable. Table 3.2 gives a sufficient value for $\mathrm{U_{lub}}$ for different numbers of tasks, but note that it may be possible to schedule a task set even if the criterion fails.

m	1	2	3	4	5	6	∞
$\mathrm{U_{lub}}$	1.000	0.828	0.780	0.757	0.743	0.735	0.690

Table 3.2: RMS schedulability criterion

In the task set from Table 3.1, the processor utilization factor is $\mathrm{U} = \sum_{i=1}^{m} \frac{C_i}{T_i} = 0.833$. The least upper bound for rate monotonic scheduling for 3 tasks is given in Table 3.2 as $\mathrm{U_{lub}} = 0.780$, and since $\mathrm{U_{lub}} < \mathrm{U}$, we cannot guarantee that this task set is schedulable.

	τ_1	τ_2	τ_3
C_i	1	2	3
T_i	4	6	8

Table 3.3: Another task set for scheduling

With the set of tasks in Table 3.3, we have that task τ_3 fails to complete within its period (8), hence this task set is not schedulable using rate monotonic scheduling. I have circled the area which fails to complete (time overflow) in Figure 3.7.

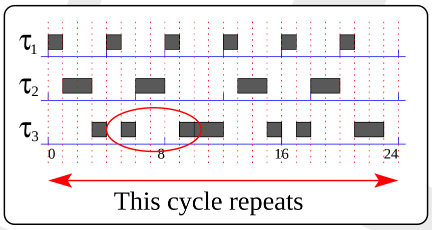


Figure 3.7: Time overflow for task set

3.2.2 EDF - Earliest deadline first scheduling

In EDF scheduling, we manage both periodic and aperiodic tasks, and those with earlier deadlines will have higher priorities. EDF is a dynamic algorithm, and is optimal for dynamic priority scheduling.

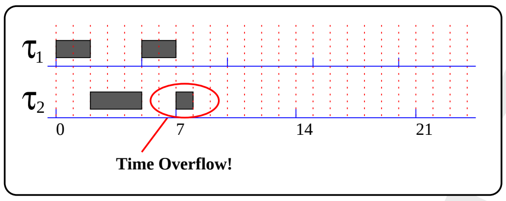
Theorem 1 *A task set is schedulable with EDF iff* $\mathrm{U} \leq 1$.

So, compared to fixed priority scheduling techniques like rate-monotonic scheduling, EDF can guarantee all the deadlines in the system at higher loading. Consider the example task set in Table 3.4. From $\mathrm{U} = 0.4 + 0.57 = 0.97 \leq 1$ we know that it is guaranteed to be schedulable under EDF.

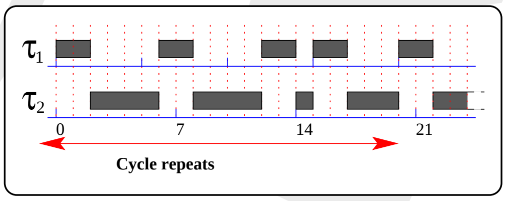
	τ_1	τ_2
C_i	2	4
T_i	5	7

Table 3.4: EDF task set for scheduling

In Figure 3.8(a) we see that it is not RMS schedulable, but EDF scheduling succeeds in Figure 3.8(b).



(a) RMS scheduling



(b) EDF scheduling

Figure 3.8: Different schedules for tasks

It is possible to calculate worst case response times of processes in EDF, dealing with other types of processes than periodic processes.

3.3 Resource access protocols

In a system with multiple tasks running on a single processor with shared resources, we need proper protocols for accessing theose shared resources. These are termed resource access protocols, and a principal method to avoid the serious problem of *priority inversion*.

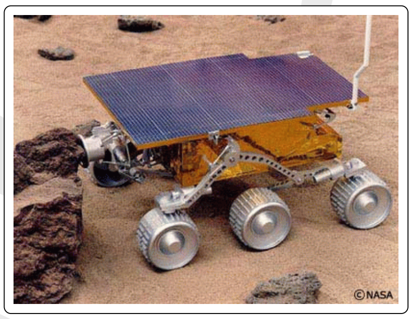


Figure 3.9: Mars rover

In 1997 the Mars pathfinder mission ran into serious problems. The spacecraft began experiencing total system resets with loss of data each time. It turned out to be due to priority inversion.

In Figure 3.10, we see a higher priority task τ_1 blocked by the much lower priority task that is holding a shared resource. The lower priority task τ_3 has acquired this resource and then been preempted by the medium priority task τ_2. In summary, τ_2 is blocking τ_1. On the pathfinder, the resource that caused this problem was a mutual exclusion semaphore.

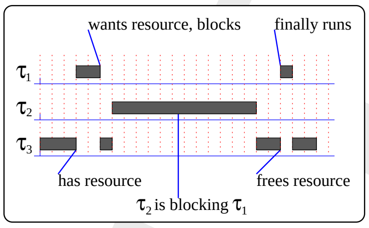


Figure 3.10: Priority inversion

So how can we avoid priority inversion? We could disallow preemption during the execution of a critical section, but this works only if critical sections are short, and might unneccesarily block higher priority processes that do not even use any shared resources. It is better to use resource access protocols such as the *priority inheritance* protocol, or the *priority ceiling* protocol.

3.3.1 Priority inheritance protocol

In the priority inheritance protocol, tasks have *nominal* and active priorities. A nominal priority is assigned by the scheduling algorithm (RMS or EDF or some other algorithm). The active priority is assigned by the protocol dynamically, specifically to avoid priority inversion.

When τ_i blocks higher-priority tasks, then its active priority is set to the highest of the priorities of the tasks it blocks. Put in another way, we could say that it temporarily inherits the highest priority of the blocked tasks. This prevents medium priority tasks from preempting τ_i and prolonging the blocking duration of the higher priority tasks. When a task is blocked on a semaphore, it transmits its active priority to the job that holds the semaphore; in general, a task inherits the highest priority of the jobs blocked by it. When a task exits a critical section, it unlocks the semaphore; the job with the highest priority that is blocked on the semaphore, if any, is awakened.

With the priority inheritance protocol, the good news is that if there are a set of distinct semaphores that can block a task τ then τ can be blocked for at most the duration of at most one critical section, one for each of the semaphores. It can never be as long as the WCET of a lower priority task. The bad news is termed *chained blocking*, where a task τ is blocked on critical sections held by lower priority jobs.

	A	B	C	D
τ_1	3	2	4	6
τ_2	4	0	6	8
τ_3	2	1	0	5

Table 3.5: Task set for resource protocols

Consider three periodic tasks τ_1, τ_2 and τ_3 (having decreasing priority) which share four resources A, B, C and D accessed using the priority inheritance protocol. The longest duration D_{iR} for the task τ_i on resource R is given by the table below. ($D_{iR} = 0$ means τ_i does not use at all the resource D). We can compute a (conservative) maximum blocking time B_i for each task like this:

$$\begin{array}{rrcl} & B_i & = & \min(B_i^\ell, B_i^s) \\ \text{and} & B_i^\ell & = & \sum_{j=i+1}^{n} \overset{\max}{k} \, [D_{j,k} : C(S_k) \geq P_i] \\ \text{and} & B_i^s & = & \sum_{k=1}^{m} \overset{\max}{j > i} \, [D_{j,k} : C(S_k) \geq P_i] \end{array}$$

where B_i^s is the sum of the durations of the longest critical sections guarded by semaphore S_k that can block τ_i, and B_i^ℓ is the sum of the durations of the longest critical sections in tasks with lower priority than τ_i, guarded by semaphore S_k, and that can block τ_i. For our example, we can use this to derive the following *blocking times*

$$\begin{array}{rcl} B_1^\ell & = & 8 + 5 = 13 \\ B_2^\ell & = & 5 \\ B_3^\ell & = & 0 \end{array} \qquad \begin{array}{rcl} B_1^s & = & 4 + 1 + 6 + 8 = 19 \\ B_2^s & = & 2 + 1 + 5 = 8 \\ B_3^s & = & 0 \end{array}$$

and so $B_1 = 13$, $B_2 = 5$ *and* $B_3 = 0$.

This calculation is reasonably efficient, but if you try to find a tighter bound, then the complexity of the algorithm would be much higher (it is exponential). A branch-and-bound algorithm (with worst-case exponential complexity) is given in Appendix A of [Raj91].

3.3.2 Priority ceiling protocol

The priority ceiling protocol is an extension of the priority inheritance protocol, which avoids chained blocking and deadlocks. The underlying idea is that a task is not allowed to enter a critical section if there are already locked semaphores which could block it eventually (due to a sub-critical section nested within the entering critical section). Hence, once a task enters a critical section, it can not be blocked by lower priority tasks till its completion.

Each semaphore S is assigned a priority ceiling $C(S)$, the priority of the highest priority task that can lock S. This is a static value. Suppose τ is currently running and it wants to lock the semaphore S. τ is allowed to lock S only if the priority of τ is strictly higher than the priority ceiling $C(S')$ of the semaphore S' where S' is the semaphore with the highest priority ceiling among all the semaphores which are currently locked by jobs other than τ. In this case, τ is said to blocked by the semaphore S' (and the job currently holding S').

When τ gets blocked by S' then the priority of τ is transmitted to the job that currently holds S'. When τ' leaves a critical section guarded by S' then it unlocks S' and the highest priority job, if any, which is blocked by S' is awakened. The priority of τ' is set to the highest priority of the job that is blocked by some semaphore that τ' is still holding. If none, the priority of τ' is set to be its nominal one.

There are two critical properties of the priority ceiling protocol:

1. a job can be blocked for at most the duration of one critical section.
2. The priority ceiling protocol prevents deadlocks.

Consider the three periodic tasks in Table 3.5. The maximum blocking time B_i for each task if the resources are accessed using the Priority Ceiling Protocol is

$$B_i = \max_{j,k} \{D_{j,k} \mid P_j < P_i, C(S_k) \geq P_i\}$$

and we have that

$$\begin{aligned} B_1 &= \max(4,2,1,6,8,5) = 8 \\ B_2 &= \max(2,1,5) = 5 \\ B_3 &= 0 \end{aligned}$$

3.4 Summary of topics

In this section, we introduced the following topics:

Task scheduling concepts. *Basic concepts of tasks in RT operating systems, feasability, constraints, timing, precedence and resource.*
Scheduling. *RMS and EDF scheduling, processor utilization factor, schedulability*
Resource access protocols. *Priority inheritance and ceiling protocols, blocking times.*